Assignment 3: LINEAR REGRESSION /OPTIMIZATION

Course Instructor: Dr. Wajahat Hussain
Prepared by: Muhammad Saifullah & Abdul Rehman Qureshi
Email: msaifullah.msee18seecs@seecs.edu.pk

Introduction

In this exercise, you will implement linear regression and get to see it work on data. Before starting on this programming exercise, we recommend clearing concepts of linear regression, gradient descent and feature normalization

Files included in this exercise

- ex1P1.ipyb – Google Colab Notebook (gradient descent) script that steps you through the exercise
- ex1P2.ipyb – Google Colab Notebook (Optimization Function) script that steps you through the exercise
- ex1data1.txt – Dataset Text file for linear regression with **one variable**

Throughout the exercise, you will be using the scripts ex1P1.ipyb and ex1P2.ipyb. These scripts set up the dataset for the problems and make calls to functions that you will write. You do not need to modify either of them. You are only required to modify functions , by following the instructions in this assignment.

For this programming exercise, you are only required to complete the first part of the exercise to implement linear regression with one variable. The second part of the exercise, which is optional, covers linear regression with multiple variables.

Where to get help:

We also strongly encourage using the online Forums and WhatsApp Group to discuss exercises with other students. **However, do not look at any source code written by others or share your source code with others.**

PART#1

1. Simple Python function

The first part of ex1P1.ipyb gives you practice with function syntax. In the the warmUpExercise, you will find a predefined function in given space to return a 5x5 or any other value identity matrix you should see output similar to the following:

```
print("Matrix a : \n", iden(5))
```

```
Matrix a :
 [[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 1.]]
```

2. Linear regression with one variable

In this part of this exercise, you will implement linear regression with one variable to predict profits for a food truck. Suppose you are the CEO of a restaurant franchise and are considering different cities for opening a new outlet. The chain already has trucks in various cities, you have data for profits and populations from the cities.

You would like to use this data to help you select which city to expand to next. The file ex1data1.txt contains the dataset for our linear regression problem. The first column is the population of a city and the second column is the profit of a food truck in that city. A negative value for profit indicates a loss.

The ex1P1.ipyb script has already been set up to load this data for you.

2.1 Plotting the Data

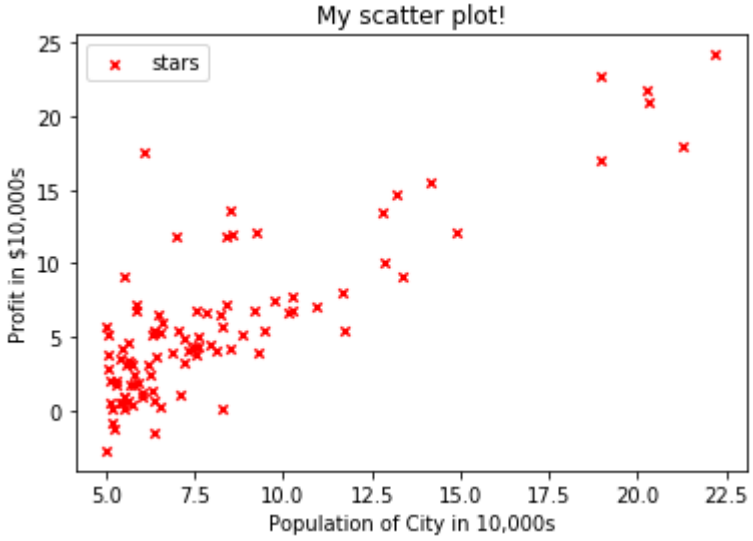
Before starting on any task, it is often useful to understand the data by visualizing it. For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population). (Many other problems that you will encounter in real life are multi-dimensional and can't be plotted on a 2-d plot.)

In ex1P1.ipyb, the dataset is loaded from the ex1data.txt into the variables X and Y:

```
# Read comma separated data
data = np.loadtxt(os.path.join('Data', path ), delimiter=',')
X, Y = data[:, 0], data[:, 1]
```

Next, the script calls the `plotdata(X,Y)` function to create a scatter plot of the data. Your job is to complete `plotdata(X,Y)` to draw the plot; fill in the following code.

Now, when you run the plotdata, our end result should look like Figure Below, with the same red “x” markers and axis labeled



LINEAR REGRESSION:

In this part, you will fit the linear regression parameters θ to our dataset using gradient descent.

Update Equations:

The objective of linear regression is to minimize the cost function

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

where the hypothesis h(x) is given by the linear model

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

Recall that the parameters of your model are the θ_j values. These are the values you will adjust to minimize cost J(θ). One way to do this is to use the batch gradient descent algorithm. In batch gradient descent, each iteration performs the update

$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j).$$

With each step of gradient descent, your parameters j come closer to the optimal values that will achieve the lowest cost J(θ). Another method is to use predefined function that optimizes or minimizes the error function.

ComputeCost:

We have already set up the data for linear regression in **ex1P1.ipyb.** In the following cell, we add another dimension to our data to accommodate the θ_0 intercept term. Do NOT execute this cell more than once.

```
m = Y.size  # number of training examples
X = np.stack([np.ones(m), X], axis=1) # it used to convert X in to (97x2), first colum is all ones
print(X.shape)
```

$$h_\theta(x) = \theta^T x = \theta_0 + \theta_1 x_1$$

```
h = np.dot(X, theta)
```

As you perform gradient descent to learn minimize the cost function J(θ) it is helpful to monitor the convergence by computing the cost. In this section, you will implement a function to calculate J(θ) so you can check the convergence of your gradient descent implementation.

Your first task is to complete the code for the function **computeCost** which computes J(θ). As you are doing this, remember that the variables X and Y are not scalar values. X is a matrix whose rows

represent the examples from the training set and Y is a vector whose each element represent the value at a given row of X.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

```
def computeCost(X,y , theta):
  m = y.size
  J = 0
  h = np.dot(X, theta)
  # ==========YOUR COST FUNCTION J HERE============


 # ===================================
  return J
```

Once you have completed the function, the next step will run `computeCost`. You will see the cost printed to the screen as given below.

```
J = computeCost(X, y, theta=np.array([0.0, 0.0]))
print('With theta = [0, 0] \nCost computed =', J)
print('Expected cost value (approximately) 32.07\n')
```

Gradient descent:

Next, you will complete a function which implements gradient descent. The loop structure has been written for you, and you only need to supply the updates to J(θ), within each iteration.

As you program, make sure you understand what you are trying to optimize and what is being updated. Keep in mind that the cost J(θ) is parameterized by the vector θ, not X and Y. That is, we minimize the value of J(θ) by changing the values of the vector θ not by changing X or Y. A good way to verify that gradient descent is working correctly is to look at the print value of J(θ) and check that it is decreasing with each step.

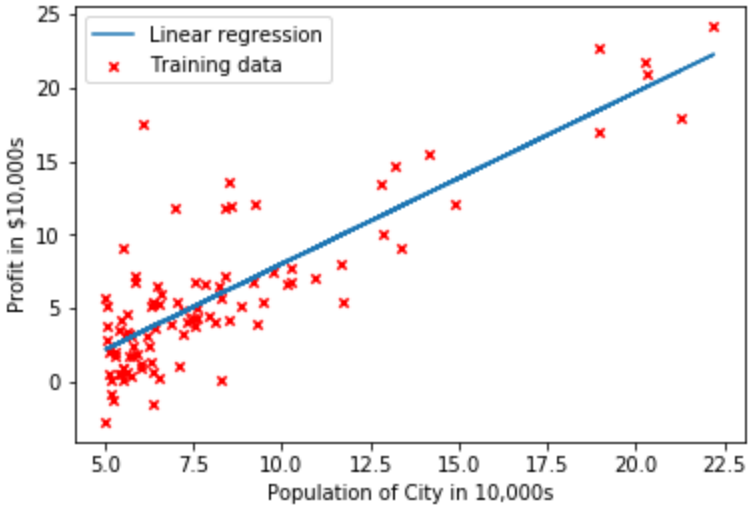
$$\theta_j := \theta_j - \alpha\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{)}.$$

The starter code for the function gradientDescent calls computeCost on every iteration and saves the cost to a python list. Assuming you have implemented gradient descent and computeCost correctly, your value of J(θ) should converge to a steady value by the end of the algorithm.

```
def gradientDescent(X, y, theta, alpha, num_iters):
  m = y.shape[0]
  theta = theta.copy()
  J_history = []
  for i in range(num_iters):
    # ==========YOUR GRADIENT DESCENT "theta" HERE============

    # ==================================
    J_history.append(computeCost(X, y, theta))


  return theta, J_history
```

Plot the training data and the linear regression:

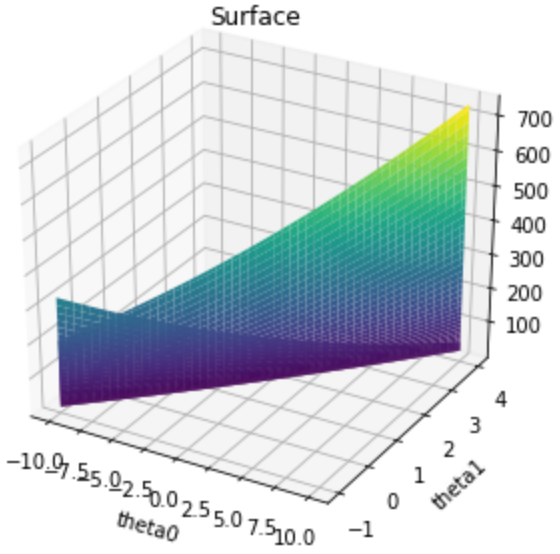
```
# initialize fitting parameters
theta = np.zeros(2)
print(theta.shape)
# some gradient descent settings
iterations = 1500
alpha = 0.01

theta, J_history = gradientDescent(X ,Y, theta, alpha, iterations)
print(J_history)


# plot the linear fit
plotdata(X[:, 1],Y)
plt.plot(X[:, 1], np.dot(X, theta))
plt.legend([ 'Linear regression','Training data',]);
plt.show()
```

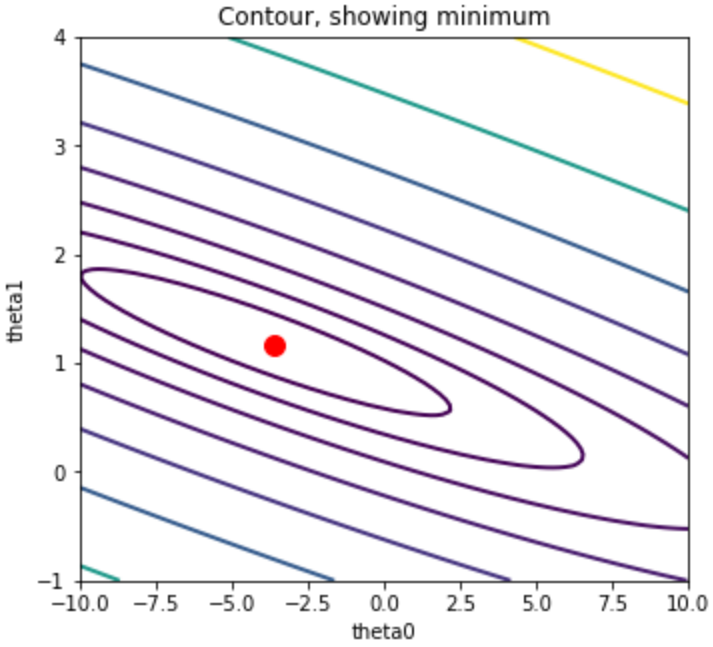


VISUALIZATION:

To understand the cost function J(θ) better, you will now plot the cost over a 2-dimensional grid of θ_0 and θ_1 values. You will not need to code anything new for this part, but you should understand how the code you have written already is creating these images.

In the next cell, the code is set up to calculate J(θ) over a grid of values using the computeCost function that you wrote. After executing the following cell, you will have a 2-D array of J(θ) values. Then, those values are used to produce surface and contour plots of J(θ) using the **matplotlib plot surface and contour functions**. The plots should look something like the following:





The purpose of these graphs is to show you how J(θ) varies with changes in θ_0 and θ_1. The cost function J(θ) is bowl-shaped and has a global minimum. (This is easier to see in the contour plot than in the 3D surface plot). This minimum is the optimal point for θ_0 and θ_1, and each step of gradient descent moves closer to this point.

FEATURE NORMALIZE:

Your task here is to complete the code in featureNormalize function:

- Subtract the mean value of each feature from the dataset.
- After subtracting the mean, additionally scale (divide) the feature values by their respective "standard deviations."

The standard deviation is a way of measuring how much variation there is in the range of values of a particular feature this is an alternative to taking the range of values (max-min). In numpy, you can use the std function to compute the standard deviation.

```
def  featureNormalize(X):

#==================YOUR CODE HERE=========

#========================================

    return X_norm, mu, sigma

X, mu, sigma = featureNormalize(X)
Y, mu, sigma = featureNormalize(Y)

X = np.stack([np.ones(m), X], axis=1)
```

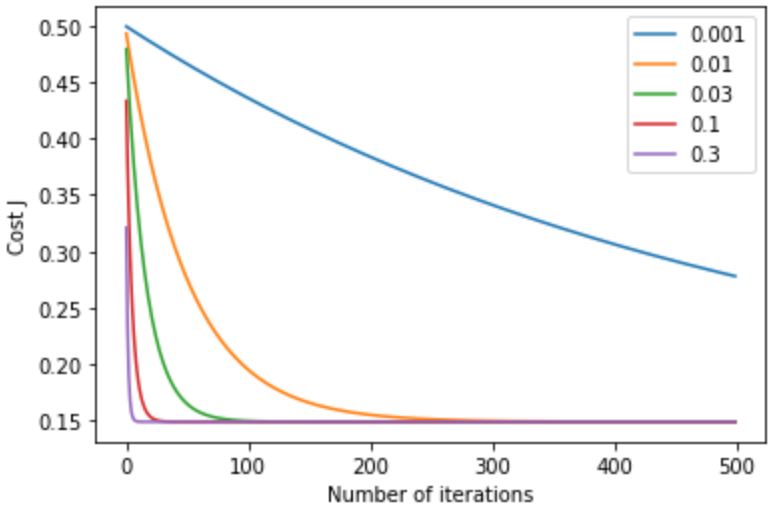
LEARNING RATES:

In this part of the exercise, you will get to try out different learning rates for the dataset and find a learning rate that converges quickly. You can change the learning rate by modifying the following code and changing the part of the code that sets the learning rate.

```
# some gradient descent settings
iterations = 500
alpha = [] #enter your learning rates
costs=[]

for i in range(5):
  theta = np.zeros(2)
  theta, J_history = gradientDescent(X ,Y, theta, alpha[i], iterations)
          # initialize fitting parameters
  costs.append(J_history)
# Plot the convergence graph

for i in range(5):
  plt.plot(np.arange(len(costs[i])), costs[i], label=str(alpha[i]))
plt.xlabel('Number of iterations')
plt.ylabel('Cost J')
plt.legend()
```

Use your implementation of gradient Descent function and run gradient descent for about 500 iterations at the chosen learning rate. The function should also return the history of J(θ) values in a vector J. After the last iteration, plot the J values against the number of the iterations.

If you picked a learning rate within a good range, your plot look similar as the following Figure.



If your graph looks very different, especially if your value of J(θ) increases or even blows up, adjust your learning rate and try again. We recommend trying values of the learning rate α (alpha) on a log-scale, at multiplicative steps of about 3 times the previous value **(0.3, 0.1, 0.03, 0.01 and 0.001).** You may also want to adjust the number of iterations you are running if that will help you see the overall trend in the curve.

PART#02

OPTIMIZATION FUNCTIONS:

We will use the **fmin and fmin_cg** function of **SciPy** in **ex1P2.ipyb**. DATA initialization will be same as above:

```
res1 = optimize.fmin_cg(J, x0, fprime=gradf,args=args)
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin_cg.html#scipy.optimize.fmin_cg

```
res2 = optimize.fmin(J, x0, args=args)
```

https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.fmin.html#scipy.optimize.fmin

Cost function minimization until it gets the optimal value, calls the gradient and finds error for each corresponding value of data.

```
#--------  COST FUNCTION-------------
def J(t,x,y):
  theta=t
  #============YOUR COST FUNCTION CODE HERE====


  #=====================================
  lr2.append(J)
  return J
```

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2$$

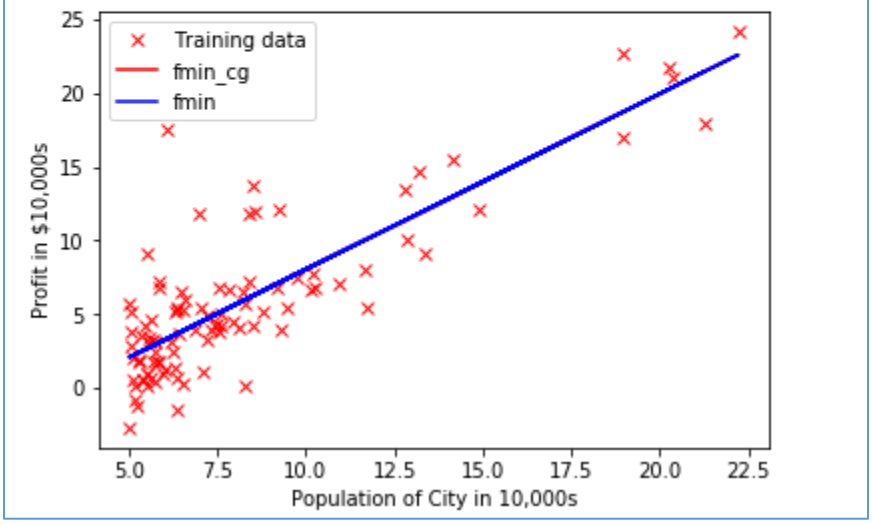
Also u have to write definition of "gradf" function for **fmin_cg** which only requires:

$$\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})x_j^{(i)} \quad \text{(simultaneously update } \theta_j \text{ for all } j).$$

```
# -----------GRADIENT ONLY FUNCTION-----------
def gradf(t, *args):
  theta =t
  #===============GRADIENT ONLY CODE HERE


 #=======================================
  return theta
```

The Final Result should resemble the figure with given above with given labels. since both optimize equally the result will be same:

```
Optimization terminated successfully.
         Current function value: 4.476971
         Iterations: 6
         Function evaluations: 14
         Gradient evaluations: 14
Optimization terminated successfully.
         Current function value: 4.476971
         Iterations: 90
         Function evaluations: 172
```



LEARNING RATES: since these function do not require alpha, single learning will be plotted by code given in the next cell.